Closing the Performance Gap between Doubles and Rationals for Octagons

Aziem Chawdhary and Andy King

University of Kent, Canterbury, CT2 7NF, UK

Abstract. Octagons have enduring appeal because their domain operations are simple, readily mapping to for-loops which apply max, min and sum to the entries of a Difference Bound Matrix (DBM). In the quest for efficiency, arithmetic is often realised with double-precision floating-point, albeit at the cost of the certainty provided by arbitrary-precision rationals. In this paper we show how Compact DBMs (CoDBMs), which have recently been proposed as a memory refinement for DBMs, enable arithmetic calculation to be short-circuited in various domain operations. We also show how comparisons can be avoided by changing the tables which underpin CoDBMs. From the perspective of implementation, the optimisations are attractive because they too are conceptually simple, following the ethos of Octagons. Yet they can halve the running time on rationals, putting CoDBMs on rationals on a par with DBMs on doubles.

1 Introduction

The dominating arithmetic operations for Difference Bound Matrices (DBMs) are addition and comparison. The speed of these operations for double-precision floating-point arithmetic is comparable with that of long integer arithmetic for modern 64-bit desktop processors, hence the trend to work with floating-point rather than idealised arithmetic, even though the latter is arguably more attractive for verification since it avoids any concerns on rounding. The problem is not just one of speed: arbitrary-precision rational numbers, as supported by the GNU multiple precision (GMP) library, require at least 24 bytes to store each entry of a DBM, whereas an IEEE 754-1983 double occupies exactly 8 bytes.

Recent progress has been made on reducing space requirements by observing the DBM entries are frequently repeated [7]. This leads to a factored representation for a DBM [7] in which the entries in the matrix are identifiers for the rationals rather than the rationals themselves. The idea is to interpret matrix entries using a table which maps each identifier to its corresponding rational; a second table is used for searching for the (unique) identifier for a given rational. The first table is used for reading a matrix and the second is used for writing to a matrix; both tables are shared across all matrices. Since the number of distinct rationals occurring as DBM entries is small, typically thousands over the lifetime of a long-running static analysis, the identifiers can be represented as 16-bit integers. Even with the overhead of the two additional tables, this reduces the space consumption of a matrix, mimicking the space savings

which come with hash consing (that incidently was invented with linear probing [12,13]). The resulting alternative representation for a DBM has been dubbed a Compact DBM (CoDBM) [7] (which is not to be confused with a Coherent DBM [23,24]). The net reduction in space over DBMs, which derives from each rational now being represented exactly once, improves cache behaviour. It also saves repeatedly initialising memory for storing the rationals, an auxiliary operation which matches the frequency of the addition and comparison. For long-running analyses, CoDBMs reduce memory consumption by approximately 30% and improve running-time by approximately 40%, by virtue of the reduction in memory initialisation and improved locality [7].

This paper focuses on the computational, rather than the space-saving aspects of CoDBMs. Our first contribution is in optimising a write to a CoDBM. A CoDBM employs an ordered table (the second table) which maps each rational encountered thus far during analysis to its unique identifier. Whenever an entry is to be written, the table is searched (using binary search) for a rational and its corresponding identifier. We show how hashing and linear probing can avoid the repeated comparisons made by binary search and avoid the need to maintain an ordered table. We report that the number of resulting comparisons (and multiplications) is indeed reduced and demonstrate a commensurate speedup and improved cache behaviour.

Our second contribution relates to join, which is one of the domain operations that occurs with high frequency. Join is computed pairwise on the entries of two DBMs, and likewise for CoDBMs, by comparing each entry point-wise and taking the maximum. Point-wise join can be simplified by checking if the two identifiers align, or if one matches the special identifier which is reserved for infinity. Both operations can be implemented in a lightweight manner using CoDBMs, thus avoiding expensive number comparison operations. These refinements constitute our second contribution.

Our third contribution exploits the infinity identifier in another domain operation: closure. Closure reduces to a sequence of addition and maximum calculations, the results of which will be infinity if either of their arguments are infinity. Thus, if an entry of the CoDBM feeds an addition, and that entry is the infinity identifier, the result of the addition is infinity, irrespective of its other argument. Likewise for maximum. A lightweight check can be introduced to detect when the inner loop of the closure calculation can be bypassed. An analogous refinement carries over to incremental closure [10,24]. These refinements make up the third contribution.

Cumulatively, these refinements close the performance gap between doubles and rationals for octagons, from which we conclude that the role of rationals needs to be reevaluated. The paper feeds into the growing body of work [2,3,7,17,26,28,29] on how best to realise octagons on stock architectures.

$$
\begin{array}{r}
x_0 \leqslant 3 \\
x_1 \leqslant 2 \\
x_0 + x_1 \leqslant 6 \\
-x_0 - x_1 \leqslant 5 \\
-x_0 \leqslant 3
\end{array}
\qquad
\begin{array}{r}
x'_0 - x'_1 \leqslant 6 \\
x'_2 - x'_3 \leqslant 4 \\
x'_0 - x'_3 \leqslant 6 \\
x'_2 - x'_1 \leqslant 6 \\
x'_1 - x'_2 \leqslant 5 \\
x'_3 - x'_0 \leqslant 5 \\
x'_1 - x'_0 \leqslant 6
\end{array}
\qquad
\begin{array}{c}
\begin{array}{cccc} x'_0 & x'_1 & x'_2 & x'_3 \end{array} \\
\begin{array}{c} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{array}
\left[\begin{array}{cccc}
\infty & 6 & \infty & 6 \\
6 & \infty & 5 & \infty \\
\infty & 6 & \infty & 4 \\
5 & \infty & \infty & \infty
\end{array}\right]
\end{array}
$$

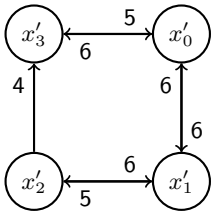

Fig. 1: Example of an octagonal system and its DBM representation

2 Background

An octagonal constraint [1,23,24] is a two-variable inequality of the syntactic form $x_i - x_j \leqslant c$, $x_i + x_j \leqslant c$ or $-x_i - x_j \leqslant c$ where c is a constant, and x_i and x_j are drawn from a finite set of variables $\{x_0, \ldots, x_{n-1}\}$. This class includes unary inequalities $x_i + x_i \leqslant c$ and $-x_i - x_i \leqslant c$ which express interval constraints. An octagon is a set of points satisfying a system of octagonal constraints. The octagon domain is the set of all octagons defined over a given set of variables.

2.1 DBMs

Implementations of the octagon domain reuse machinery developed for solving difference constraints of the form $x_i - x_j \leqslant c$. An octagonal constraint over $\{x_0, \ldots, x_{n-1}\}$ can be translated [24] to a difference constraint over an augmented set of variables $\{x'_0, \ldots, x'_{2n-1}\}$, which are interpreted by $x'_{2i} = x_i$ and $x'_{2i+1} = -x_i$. The translation proceeds as follows:

$$
\begin{array}{rl}
x_i - x_j \leqslant c \rightsquigarrow & x'_{2i} - x'_{2j} \leqslant c \quad \wedge \; x'_{2j+1} - x'_{2i+1} \leqslant c \\
x_i + x_j \leqslant c \rightsquigarrow & x'_{2i} - x'_{2j+1} \leqslant c \quad \wedge \quad x'_{2j} - x'_{2i+1} \leqslant c \\
-x_i - x_j \leqslant c \rightsquigarrow & x'_{2i+1} - x'_{2j} \leqslant c \quad \wedge \quad x'_{2j+1} - x'_{2i} \leqslant c \\
x_i \leqslant c \rightsquigarrow & x'_{2i} - x'_{2i+1} \leqslant 2c \\
-x_i \leqslant c \rightsquigarrow & x'_{2i+1} - x'_{2i} \leqslant 2c
\end{array}
$$

A difference bound matrix (DBM) [11,22] (denoted $\mathbf{m}$) which is a square matrix of dimension $n \times n$, is commonly used to represent a systems of n^2 (syntactically irredundant [21]) difference constraints over n variables. The entry $\mathbf{m}_{i,j}$ represents the constant c of the inequality $x_i - x_j \leqslant c$ where $i, j \in [0, n)$. Since an octagonal constraint system over n variables translates to a difference constraint system over $2n$ variables, a DBM representing an octagon has dimension $2n \times 2n$. Figure 1 illustrates how an octagon translates to a system of differences. The entries of the DBM correspond to the constants in the difference constraints. Note how differences which are (syntactically) absent from the system lead to entries which take a symbolic value of ∞. Observe how the DBM can be viewed as an adjacency matrix for the illustrated graph.

```
1: function CLOSE(m)
2:     for k ∈ {0, ..., 2n − 1} do
3:         for i ∈ {0, ..., 2n − 1} do
4:             for j ∈ {0, ..., 2n − 1} do
5:                 m_{i,j} ← min(m_{i,j}, m_{i,k} + m_{k,j})
6:             end for
7:         end for
8:     end for
9:     return m
10: end function
```

```
1: function STR(m)
2:     for i ∈ {0, ..., 2n − 1} do
3:         for j ∈ {0, ..., 2n − 1} do
4:             m_{i,j} ← min(m_{i,j}, (m_{i,ī} + m_{j̄,j})/2)
5:         end for
6:     end for
7:     return m
8: end function
```

Fig. 2: Non-incremental closure and strengthening

2.2 Closure

Closure properties define canonical representations of DBMs, and can decide satisfiability and support operations such as join and projection. Bellman [4] showed that the satisfiability of a difference system can be decided using shortest path algorithms on a graph representing the differences. If the graph contains a negative cycle (a cycle whose edge weights sum to a negative value) then the difference system is unsatisfiable. The same applies for DBMs representing octagons. Closure propagates all the implicit (entailed) constraints in a system, leaving each entry in the DBM with the sharpest possible constraint entailed between the variables. A DBM $\mathbf{m}$ of dimension $n \times n$ is said to be closed iff $\forall i.\mathbf{m}_{i,i} = 0$ for all $i \in [0, n)$ and $\mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,k} + \mathbf{m}_{k,j}$ for all $i, j, k \in [0, n)$. Zero diagonal elements are enjoyed by octagons which are satisfiable. The DBM is said to be strongly closed iff additionally $\forall i, j.\mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,\bar{i}}/2 + \mathbf{m}_{\bar{j},j}/2$ for all $i, j \in [0, n)$, where $\bar{i}$ is $i + 1$ if i is even, and $i - 1$ otherwise. Strong closure merges a pair of unary constraints into a single binary constraint: the binary constraint $2(x'_i - x'_j) \leqslant \mathbf{m}_{i,\bar{i}} + \mathbf{m}_{\bar{j},j}$ following from the two unary constraints $2x'_i = x'_i - x'_{\bar{i}} \leqslant \mathbf{m}_{i,\bar{i}}$ and $-2x'_j = x'_{\bar{j}} - x'_j \leqslant \mathbf{m}_{\bar{j},j}$. Figure 2 gives a cubic implementation which tightens a DBM to ensure closure and a quadratic pass which enforces strong closure. Satisfiability is checked by merely inspecting the diagonal of the tightened DBM.

2.3 Incremental Closure

Minè introduced incremental closure [24] which reestablishes closure once a small number of constraints are added to a closed DBM. This algorithm was

```
1: function IncClose(m, x'_a − x'_b ⩽ d)
2:     for i ∈ {0, ..., 2n − 1} do
3:         for j ∈ {0, ..., 2n − 1} do
4:             m'_{i,j} ← min( m_{i,j},
                               m_{i,a} + d + m_{b,j},
                               m_{i,b̄} + d + m_{ā,j},
                               m_{i,b̄} + d + m_{ā,a} + d + m_{b,j},
                               m_{i,a} + d + m_{b,b̄} + d + m_{ā,j} )
5:         end for
6:     end for
7: end function
```

```
1: function IncCloseHoist(m, x'_a − x'_b ⩽ d)
2:     t1 ← d + m_{ā,a} + d;
3:     t2 ← d + m_{b,b̄} + d;
4:     for i ∈ {0, ..., 2n − 1} do
5:         t3 ← min(m_{i,a} + d, m_{i,b̄} + t1);
6:         t4 ← min(m_{i,b̄} + d, m_{i,a} + t2);
7:         for j ∈ {0, ..., 2n − 1} do
8:             m_{i,j} ← min(m_{i,j}, t3 + m_{b,j}, t4 + m_{ā,j})
9:         end for
10:    end for
11:    return m
12: end function
```

Fig. 3: Incremental Closure (without and with code hoisting)

```
1: function Join(m^1, m^2)
2:     for i ∈ {0, ..., 2n − 1} do
3:         for j ∈ {0, ..., 2n − 1} do
4:             m_{i,j} ← max(m^1_{i,j}, m^2_{i,j})
5:         end for
6:     end for
7:     return m
8: end function
```

Fig. 4: Join of two closed DBMs

subsequently refined [9,10] to give the quadratic algorithm listed in figure 3, presented both with and without loop-invariant code hoisting. The idea is to determine how each DBM entry $\mathbf{m}_{i,j}$ is effected by the addition of a new constraint $x'_a - x'_b \leqslant d$, independent of every other DBM entry.

The force of (strong) closure, whether incremental or not, is that it gives a canonical representation for DBMs; it also reduces join to the pointwise max of two closed DBMs, to give the quadratic join operation illustrated in Figure 4.

2.4 Apron

Apron is a widely-used Octagon domain library [18] which is implemented in C, with bindings for C++, Java and OCaml. It supports various number systems. Numbers are represented by a type `bound_t`, which, depending on compile-time

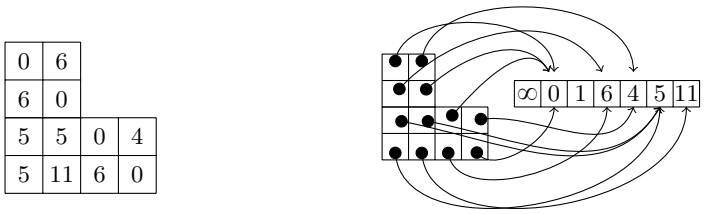

Fig. 5: Example illustrating the difference between DBMs and CoDBMs

```
1: function SEARCH(v)
2:     if v ∈ dom(search) then
3:         return search(v)
4:     else
5:         values ← values ∪ {ℓ ↦ v} where ℓ ∉ dom(values)
6:         search ← search ∪ {v ↦ ℓ}
7:         return ℓ
8:     end if
9: end function
```

Fig. 6: Searching and extending search and values

options, will select a specific header file with a specific concrete implementation of numbers extended with symbolic values of $-\infty$ and $+\infty$. Every `bound_t` object is initialised via a call to `bound_init`, which in the case of GMP rationals will call a `malloc` function. Numbers of type `bound_t` cannot be assigned directly, but instead are assigned via function calls such as `bound_set`.

DBMs are stored by taking advantage of coherence [24], which can be assumed without loss of generality. A DBM $\mathbf{m}$ is said to be coherent if $\mathbf{m}_{i,j} = \mathbf{m}_{\bar{j},\bar{\imath}}$ for all $i, j \in [0, n)$. Coherence allows a half-matrix to be represented which, in turn, can then be packed into a (linear) array of `bound_t` objects as follows: If $i \geqslant j$ or $i = \bar{j}$ then the entry at (i, j) in the DBM is stored at index $j + \lfloor i^2/2 \rfloor$ in the array. Otherwise (i, j) is stored at the index location reserved for entry $(\bar{j}, \bar{\imath})$. A DBM of dimension $n \times n$ then requires an array of size $2n(n + 1)$.

2.5 CoDBMs

Compact DBMs (CoDBMs) [7] redistribute the cost of memory allocation and initialisation, and do so in a way that is sensitive to the relative frequency of DBM reads to DBM writes (the latter being less frequent than the former). CoDBMs are matrices where the entries are identifiers (short integers), rather than numeric values (rationals), and each identifier references a number in a shared number pool, as illustrated in figure 5. The number pool is abstracted by two functions: $\mathsf{values} : \mathbb{N} \to \mathbb{Q}$ and $\mathsf{search} : \mathbb{Q} \to \mathbb{N}$, which are mutual inverses.

The change from DBMs to CoDBMs requires a new API for reading and writing an entry $\mathbf{c}_{i,j}$ of a CoDBM $\mathbf{c}$. Reading $\mathbf{c}_{i,j}$ amounts to interpreting the index stored in $\mathbf{c}_{i,j}$ using values to obtain a value $v = \mathsf{values}(\mathbf{c}_{i,j})$. Writing

a value v to $\mathbf{c}_{i,j}$ involves applying a function $\ell = \textsc{search}(v)$ to retrieve the identifier ℓ for v and then assigning $\mathbf{c}_{i,j}$ to ℓ. The function Search, which is listed in figure 6, manufactures a unique identifier if v is fresh and extends values and search accordingly. Previous work [7] realised values as an array of rationals and search as an ordered array of rationals, the index of a particular rational defining the identifier. The identifier was found using Bisection search [32]. CoDBMs achieve speedups because they store identifiers which are more compact than rationals (improving locality) and each distinct rational is stored once in the number pool (saving initialisation).

3 Hashing

The GMP manual [15] alludes to the fact that comparisons on rationals are expensive since $p/q \leqslant r/s$ reduces to $sp \leqslant qr$ if the denominators p and s are positive. Comparison thus involves two multiplications in general. Moreover, Search is invoked on every write to the CoDBM and each invocation will compute $\lceil log_2(n) \rceil$ comparisons in the worst case where n is the number of rationals in number pool. Thus, even if the pool contains just 256 rationals, a write can induce 16 multi-precision multiplications. Moreover, to insert a new rational into an ordered table it is necessary to shuffle along other elements. These costs motivate hashing.

A rational r is hashed by converting it to a double-precision floating point number f, an operation which is supported by GMP. If s is the size of the hash table then a multiplicative hash [20] $h(f)$ is computed by calculating $\ell = \lfloor fs(1 + \sqrt{5})/2 \rfloor \mod s$ with floating-point arithmetic and defining $h(f) = \ell$ if $\ell \geqslant 0$ and $h(f) = s - \ell$ otherwise. Hashing with the Golden Ratio helps ensure that the hashes are scattered evenly, reducing the chance of collisions [20, Chapter 6.4]. If a rational does not exist at entry $h(f)$ then r is inserted at this entry and $h(f)$ is returned by Search. If the entry $h(f)$ is already occupied by a rational r', then an equality check $r = r'$ is performed on rationals (which is constant-time by virtue of a canonical representation). If $r = r'$ succeeds then $h(f)$ is returned by Search. If $r = r'$ fails then linear probing is applied to find the next consecutive (modulo s) identifier ℓ' whose entry is empty in the table. The rational r is then inserted at ℓ' and ℓ' is returned by Search. Note if that r is exceptionally large then f can conceivably be NaN in which case a constant hash can be assigned.

Although multiplicative hashes are not renowned for avoiding collisions, they are simple, and it turns out that collisions are incredibly rare because the number of distinct rationals is small and thus the occupancy of the table is low.

4 Optimising Join

DBMs typically contain many symbolic infinity values: a property has sparked an interest in using sparse representations for difference constraints [14]. However, sparse representations complicate the join of octagons [19], the simplicity of

```
1: function SETDBMMAX(c, (i, j), c¹, c²)
2:     BOUND_INIT(TMP)
3:     TMP ← BOUND_MAX(values(c¹_{i,j}), values(c²_{i,j}))
4:     c_{i,j} ← SEARCH(TMP)
5: end function
```

```
1: function SETDBMMAX_OPT(c, (i, j), c¹, c²)
2:     if (c¹_{i,j} = ℓ∞ ∨ c²_{i,j} = ℓ∞ then
3:         c_{i,j} ← ℓ∞
4:     else if (c¹_{i,j} = c²_{i,j}) then
5:         c_{i,j} ← c¹_{i,j}
6:     else
7:         if values(c¹_{i,j}) ⩾ values(c²_{i,j}) then
8:             c_{i,j} ← c¹_{i,j}
9:         else
10:            c_{i,j} ← c²_{i,j}
11:        end if
12:    end if
13: end function
```

Fig. 7: DBM max operation used in join and widening, and its optimised version

which we want to preserve. Nevertheless, the identifiers employed by CoDBMs enable join to bypass vacuous DBM entries, without adding any conceptual complexity to join itself. The idea is to merely fix the identifier for symbolic infinity up-front so that infinity can be intercepted with a lightweight check without inspecting the symbolic value itself.

To reflect on the cost of join, consider the implementation of a max operation (SETDBMMAX) used in join, which assigns $\mathbf{c}_{i,j}$ to the maximum of $\mathbf{c^1}_{i,j}$ and $\mathbf{c^2}_{i,j}$ shown in figure 7. Quite apart from the two multi-precision multiplications used in the comparison which underpins BOUND_MAX in line 3, line 2 allocates and initialises memory (which we make explicit to highlight a hidden cost).

Yet if either $\mathbf{c^1}_{i,j}$ or $\mathbf{c^2}_{i,j}$ is the identifier for infinity, denoted ℓ_∞, then there is no need to perform any comparison between rationals: the entry $\mathbf{c}_{i,j}$ can simple be assigned the identifier ℓ_∞. Moreover, if the identifiers $\mathbf{c^1}_{i,j}$ and $\mathbf{c^2}_{i,j}$ align, then again a rational comparison is not needed. In fact, only in exceptional cases do the rationals need to be looked-up at all, which reduces memory pressure. This optimisation can be rolled out for widening which also uses SETDBMMAX. An analogous optimisation applies for meet, using min instead of max (though meet arises relatively infrequently during analysis).

5 Optimising Closure

Figure 8 shows how the identifier ℓ_∞ can be likewise trapped to speed up non-incremental and incremental closure. The observation is that if $\mathbf{c}_{i,k} = \ell_\infty$ then then sum $\mathsf{values}(\mathbf{c}_{i,k}) + \mathsf{values}(\mathbf{c}_{k,j})$ will be infinity irrespective of the identifier stored in $\mathbf{c}_{k,j}$. Moreover, the check $\mathbf{c}_{i,k} = \ell_\infty$ is performed on identifiers (integers) rather than rationals, so has negligible overhead, yet it potentially enables the entire inner loop of closure to be short-circuited (see CLOSEOPT of figure 8). Incremental closure algorithm can also be optimised (see INCCLOSEHOISTOPT

```
1: function CloseOpt(c)
2:     for k ∈ {0, ..., 2n − 1} do
3:         for i ∈ {0, ..., 2n − 1} do
4:             if c_{i,k} ≠ ℓ_∞ then
5:                 for j ∈ {0, ..., 2n − 1} do
6:                     c_{i,j} ← Search(min(values(c_{i,j}), values(c_{i,k}) + values(c_{k,j})))
7:                 end for
8:             end if
9:         end for
10:    end for
11: end function
12:
13: function IncCloseHoistOpt(c, x'_a − x'_b ⩽ d)
14:    t_1 ← d + values(c_{ā,a}) + d;
15:    t_2 ← d + values(c_{b,b̄}) + d;
16:    for i ∈ {0, ..., 2n − 1} do
17:        if c_{i,a} ≠ ℓ_∞ ∧ c_{i,b̄} ≠ ℓ_∞ then
18:            t_3 ← min(values(c_{i,a}) + d, values(c_{i,b̄}) + t_1);
19:            t_4 ← min(values(c_{i,b̄}) + d, values(c_{i,a}) + t_2);
20:            for j ∈ {0, ..., 2n − 1} do
21:                if c_{b,j} ≠ ℓ_∞ ∧ c_{ā,j} ≠ ℓ_∞ then
22:                    c_{i,j} ← Search(min(values(c_{i,j}), t_3 + values(c_{b,j}), t_4 + values(c_{ā,j})))
23:                end if
24:            end for
25:        end if
26:    end for
27: end function
```

Fig. 8: Optimised versions of closure and incremental closure

of figure 8) where both the outer and inner loop can be skipped if certain indices match the fixed identifier ℓ_∞. These optimisations will only really benefit closure calculations on large CoDBMs so it is important that the checks are sufficiently lightweight to not overburden closures operating on small CoDBMs.

6 Experiments

This section compares the performance of CoDBMs over rationals against DBMs over doubles using three abstract interpreters [6,16,30], reporting execution times augmented with memory statistics for the longest running analyses. All statistics were gathered on a Linux machine equipped with 128GB of RAM and dual 2.0GHz Intel Xeon E5-2650 processors. Timings were averaged over five runs using multitime (`http://tratt.net/laurie/src/multitime/`) and include the time required to perform a complete analysis from parsing source to output.

6.1 FuncTion: Timings

Figure 9 presents the running times of the FuncTion termination analyser on all the 58 benchmarks from its repository (`https://github.com/caterinaurban/function`); timings which are fully detailed in the accompanying technical report [8]. FuncTion [30] applies abstract interpretation to infer piece-wise ranking

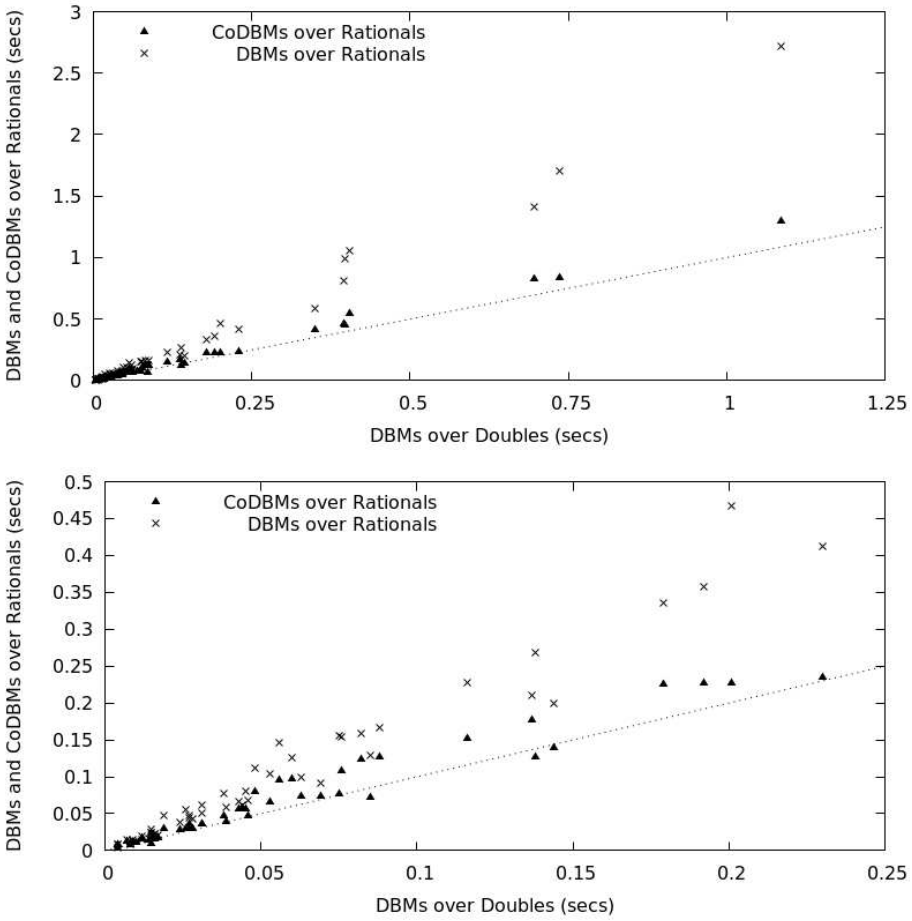

Fig. 9: CoDBMs and DBMs for rationals against DBMs for doubles

functions for verifying termination. It is implemented in OCaml and can analyse simple programs in a C-like language. FuncTion has options for intervals, arbitrary polyhedra and octagons. For octagons, the default setting is Apron DBMs instantiated with rationals, reflecting a focus on verification. Doubles and CoDBMs were supported by changing the build system.

The cross marks of the scatter plot compare the running time for Apron DBMs over rationals against the execution time for Apron DBMs over doubles, so as quantify the overhead induced by rationals. The dotted-line has the gradient of one. The triangles illustrate the execution time of CoDBMs over rationals, equipped with the complete set of optimisations, again relative to doubles on DBMs. The top graph illustrates these timing for all benchmarks, whereas the

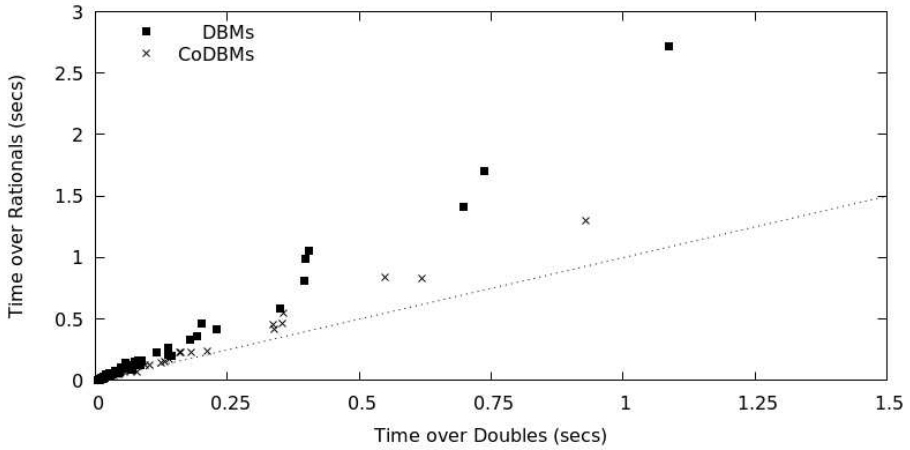

Fig. 10: DBMs for rationals against DBMs for doubles and likewise for CoDBMs

bottom graph zooms in on the cluster of benchmarks around the origin. The proximity of the triangles near to the dotted line suggests that CoDBMs over rationals, when optimised, compare favourably to DBMs over doubles, at least for the benchmarks under test. Figure 10 compares CoDBMs to DBMs using a different perspective: it also compares CoDBMs over rationals to CoDBMs over doubles, showing how CoDBMs, when optimised, reduce the overhead of moving from doubles to rationals relative to the same move on DBMs.

6.2 Crab-LLVM: Timings

To compare rationals against doubles with a state-of-the-art [16] inter-procedural analysis, Crab-LLVM (`https://github.com/seahorn/crab-llvm`) was built against Apron and CoDBMs and then applied to the 596 benchmarks of product-line SV-COMP series to infer octagonal invariants. This setup also exercised the domain operations from C rather than through OCaml bindings so as to check whether the bindings impact on performance.

The large number of benchmarks make a scatter plot infeasible, hence Figure 11 plots the cumulative running time for the first n benchmarks against n itself; the technical report [8] details all these timings. These benchmarks divide into the elevator, email, and minepump sub-series of unreachability problems. Each benchmark in each sub-series has a broadly similar execution time, so the cumulative running time approximates a piece-wise linear function. The headline message is that, again, CoDBMs over rationals approach the performance of DBMS over doubles; moreover CoDBMs provide a modest gain on DBMs for doubles when all the optimisations are in place. Interestingly, Crab-LLVM defaults to the Elina library [29] which partitions a DBM on-the-fly into sub-DBMs

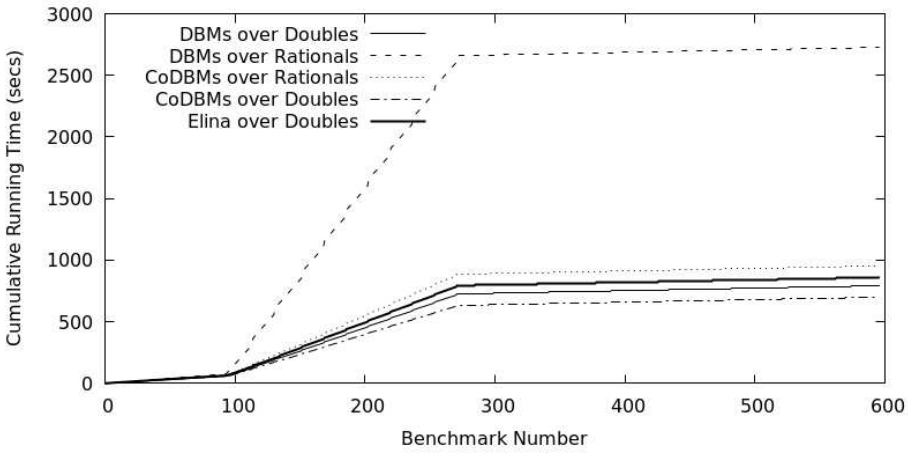

Fig. 11: Cumulative execution times over product-lines SV-COMP benchmarks

Abbrv	Benchmark	LOC	Description
lev	levenstein	187	Levenstein string distance library
sol	solitaire	334	card cipher
2048	2048	435	2048 game
kh	khash	652	hash code from klib C library
taes	Tiny-AES	813	portable AES-128 implementation
mod	libmodbus	7685	library to interact with Modbus protocol
mgmp	mini-gmp	11787	subset of GMP library
bzip	bzip-single-file	74017	bzip single file for static analysis benchmarking

Fig. 12: Benchmarks

that do not share variables, whilst simultaneously applying vectorisation. This invited a comparison. Elina did not perform as well as CoDBMs with the join and closure optimisations or even DBMs. Though unexpected, we include these results nevertheless. (It should be stressed that Elina was built exactly as specified, to the same level of optimisation as DBMs and CoDBMs, with the vector flag correctly set for an E5-2650 which supports vectorisation. Elina currently does not provide OCaml bindings otherwise we would have performed further comparisons using FuncTion and Frama-C.)

6.3 Frama-C: Timings

With an eye towards longer running analyses, EVA [6], the abstract interpretation plugin for Frama-C Sulfur, was used for comparing rationals against doubles for DBMs and CoDBMs. EVA is a prototype analyser for C99 which supports Apron but does not provide state-of-the-art optimisations such as au-

	Apron DBM			CoDBM rationals				CoDBM doubles			
Abbrv	rationals	doubles	Ids	Bisect	Hash	Join	Close	Bisect	Hash	Join	Close
lev	22.78	4.71	900	12.16	10.66	8.15	7.41	5.87	4.78	4.74	4.30
sol	92.13	42.77	2161	80.22	71.00	51.48	50.61	52.09	44.03	42.89	42.52
2048	37.74	8.28	358	22.36	19.57	14.09	13.31	10.79	8.73	8.26	7.77
kh	3.087	1.92	196	2.44	2.447	2.167	2.131	1.871	2.014	1.928	1.843
taes	1883.50	153.43	140	740.47	663.37	411.72	386.67	261.32	164.85	143.63	129.67
mod	820.88	96.68	3627	558.27	494.12	321.72	293.68	192.63	111.92	102.78	91.70
mgmp	4.33	4.36	126	4.33	4.28	4.18	4.16	4.38	4.28	4.22	4.15
bzip	655.82	54.15	262	232.63	94.98	95.49	94.20	168.90	60.01	59.00	58.99

Fig. 13: Frama-C EVA plugin timings

tomatic variable clustering [17] or access-based localisation [3]. Nevertheless, figure 12 lists the programs used for benchmarking, which represent eight programs from the Frama-C case study repository (`https://github.com/Frama-C/open-source-case-studies`) that successfully terminate when the EVA plugin is instantiated with octagons.

Figure 13 details the overall execution for DBMs, both for doubles and rationals. Interestingly, teas, mod and bzip are ten-fold slower with rationals than doubles for DBMs. This stems from a high number of DBMs with high dimension so that, cumulatively, the total number of DBMs entries created during analysis for each of these three problems is between 40- and 400-fold the number of DBM entries created for any of the other five problems. The Ids column records the total number of identifiers (distinct DBM entries) used over the lifetime of each analysis. These counts are significantly smaller than the total number of DBM entries over the lifetime of each analysis by typically six orders of magnitude larger. (Shorter running analyses typically have smaller Ids counts.)

The Bisect column records the overall running time when bisection search is used to locate an identifier. Hash gives the runtime when hashing and linear probing is used instead. The hash table was allocated to store 10K rationals and collisions were barely discernible (since the Ids count was always low) hence the table was never expanded. Join presents the time when hashing is augmented with join (and meet) optimisation. The Close column additions applies the optimisations on both closure and incremental closure. For the longer running problems, Hash significantly improves on Bisect and Join significantly improves on Bisect (which the notable exception of mgmp where it has little effect). Close makes a less significant improvement on Join, but is useful nevertheless.

As a control, the last four columns of the table repeat the experiments but with CoDBMs instantiated with doubles. Since arithmetic is faster on doubles and doubles are compact, it is surprising to see that CoDBMs sometimes outperform DBMs. We surmise that the speedup comes from reduced memory pressure.

		CoDBM			
rationals	Apron DBM	Bisect	Hash	Join	Close
lev	1,498,168	1,457,632	1,452,868	106,072	106,620
sol	6,187,568	10,718,464	10,717,924	246,008	247,044
2048	3,714,408	2,999,468	3,034,068	170,840	170,700
kh	163,840	142,168	142,264	69,696	69,440
taes	20,845,632	119,190,024	119,153,636	591,228	590,700
mod	28,945,580	73,116,452	73,111,892	911,572	901,788
mgmp	89,496	89,992	88,844	89,308	88,508
bzip	9,499,460	1,636,276	1,634,444	551,664	551,516
		CoDBM			
doubles	Apron DBM	Bisect	Hash	Join	Close
lev	191,968	106,264	106,004	106,200	105,756
sol	690,476	284,556	284,068	283,812	283,952
2048	413,916	167,432	165,548	165,712	165,284
kh	73,608	68,780	68,720	68,376	68,696
taes	2,044,832	603,056	604,440	604,416	604,420
mod	2,871,612	921,780	939,056	919,808	922,108
mgmp	89,832	88,092	88,548	88,008	88,628
bzip	1,158,460	555,216	551,884	552,524	552,988

Fig. 14: Memory Usage in kb for Frama-C: rationals above and doubles below

6.4 Frama-C: Memory Usage

Figure 14 records total memory usage as harvested by GNU time, which returns the maximum resident set size of the process during its lifetime. The table contains some surprises. First, memory usage for CoDBMs over rationals gives a net increase on DBMs over rationals for some problems. This increase occurs for the Sulfur version of Frama-C; the previous version gives a consistent reduction in memory usage (which is expressed as a percentage in the same column). The problem in recycling memory seems to relate to GNU multi-precision arithmetic since Sulfur gives a reduction when the same CoDBM code is instantiated with doubles. The second surprise is that Sulfur gives a dramatic overall decrease when CoDBMs are deployed with the join optimisation. This stems from the allocation and initialisation space for each maxima. This only occurs for rationals, hence Join offers little improvement for doubles.

Figure 15 records cache statistics for DBMs and CoDBMs which were harvested with Cachegrind [25] (for four representative benchmarks). Refs and Miss respectively denote the number of memory references and last-level cache misses for rationals for both DBMs and CoDBMs, where m denotes millions. A, B, H, J and C abbreviate the column names of Figure 14. Reading an element from a CoDBM incurs an extra layer of indirection compared to a DBM and writing to a CoDBM can incur multiple memory references, so one might expect additional memory references. Yet the number of references reduces uniformally between DBMs and CoDBMs for Bisect and then across the CoDBM optimisations. The

Abbrv		Insts	Refs	Miss	Rate	Abbrv		Insts	Refs	Miss	Rate
2048	A	242190m	99227m	185m	0.187	lev	A	145649m	59428m	72m	0.121
2048	B	125304m	47092m	52m	0.112	lev	B	67952m	25346m	25m	0.101
2048	H	95078m	41082m	52m	0.128	lev	H	52800m	22565m	25m	0.113
2048	J	65529m	30433m	5m	0.016	lev	J	38634m	17472m	3m	0.017
2048	C	56981m	26670m	5m	0.018	lev	C	32925m	14975m	3m	0.020
kh	A	13870m	6545m	5.2m	0.08	bzip	A	52677557m	2166803m	19579m	0.904
kh	B	9228m	4595m	2.7m	0.06	bzip	B	1855916m	410861m	76m	0.019
kh	H	8579m	4465m	2.7m	0.06	bzip	H	533543m	229553m	76m	0.033
kh	J	7840m	4196m	1.3m	0.03	bzip	J	522984m	225737m	57m	0.026
kh	C	7690m	4130m	1.3m	0.03	bzip	C	512926m	221316m	57m	0.026

Fig. 15: Instruction count and cache statistics for Frama-C for rationals

number of cache misses reduces even faster, indicating that locality is improved too, hence the decreasing cache miss-rate percentage (Rate). Reassuringly, the number of misses does not increase between Bisect and Hash, even though hashing can map a number to any location in the hash table, whereas bisection will only search the portion of the second table which is actually populated. A single (non-local) read into the hash table (which is the norm as collisions are rare) seems to more than offset the multiple reads incurred by bisection, which become progressively more local as search proceeds. Join gives an order of magnitude reduction in the number of misses because it bypasses accessing numbers in the first table as well as avoiding initialising a temporary variable and then storing the maxima. Cachegrind also records the number of instructions executed, which is a reflected in the Insts column, and is a proxy for work. The reduction in instruction count stands independent of the timings which are ultimately dependent on system behaviour.

7 Related Work

The tension between the elegance of octagons and their scalability has motivated a number of imaginative techniques [2,3,5,7,17,26,28,29] for enhancing octagonal analysis. First, variable clustering was proposed [5,23,31] for grouping variables into sets which scope the relationships that are tracked. However, deciding variable groupings is an art, although there has been recent progress made in automating decomposition both before [17] and during [29] analysis.

Second, the domain operations themselves have been refined, notably showing how strengthening (the act of combining pairs of unary octagon constraints to improve binary octagon constraints) need not be applied repeatedly, but instead can be left to a single post-processing step [1]. This led to a significant performance improvement of approximately 20% [1].

Thirdly, and more recently, there has been a move to curb the size of DBMs using sparse analyses [27] and access-based localisation techniques [3]. Access-

based localisation uses scoping heuristics to adjust the size of the DBM to those variables that can actually be updated [3]. Sparse analyses generalise access-based localisation techniques, using data dependencies to adjust the size of abstract states propagated to method calls: [27] defines a generic technique to apply sparse techniques to abstract interpretation and combines this with variable packing to scale an octagon-based abstract interpreter for C programs. Access-based localisation and sparse frameworks (and variable clustering too) are orthogonal to our work, and can take advantage of the techniques introduced in this paper. Sparse matrix representations have been proposed for octagons [19] and differences [14] as an alternative to DBMs, but these representations sit at odds with the simplicity of the algorithms originally proposed for the domain. The desirable property of strong closure [24] (the normal form for octagons) does not hold for a sparse representation, motivating the need to rework domain operations [19].

Fourthly, there has been a move to better exploit the underlying architecture, either to harness GPUs [2] or advanced vector extensions (AVX) [29] in closure and strengthening (the latter being the first work to comment on the impact of cache misses in domain engineering).

8 Conclusion Discussions

The paper contributes to the growing body of work on domain engineering, where performance is improved, often in small steps, by devising refinements which relieve the pressure of the most commonly occuring domain operations. We buck the trend towards instantiating octagons with doubles by showing how CoDBMs, which save space over DBMs, can also save on computation if equipped with simple optimisations. These optimisations enable arithmetic to be short-circuited in join and closure, and also avoid repeated comparisons by changing the tables which underpin CoDBMs. The net effect is to put rationals on a par with doubles, so as to simultaneously achieve performance and soundness.

In terms of future work, it would be interesting to apply caching more aggressively and determine whether the hash could preserve the ordering on the rationals. If so, then join (and meet) could be further refined by comparing hashes. Moreover, it would be interesting to investigate temporal locality on rational arithmetic itself and determine if caching could be deployed to further accelerate the domain operations.

Acknowledgements We thank Colin King at Canonical and Laurie Tratt at Kings for their help with the performance analysis which underpinned this work. We thank Oleg Kiselyov for his help navigating soviet computer science literature. This work was funded by EPSRC EP/K032585/1 and EPSRC EP/N020243/1.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational Shapes for Numeric Abstractions: Improved Algorithms and Proofs of Correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.
2. F. Banterle and R. Giacobazzi. A Fast Implementation of the Octagon Abstract Domain on Graphics Hardware. In *SAS*, volume 4634 of *LNCS*, pages 315–335. Springer, 2007.
3. E. Beckschulze, S. Kowalewski, and J. Brauer. Access-Based Localization for Octagons. *Electronic Notes in Theoretical Computer Science*, 287:29–40, 2012.
4. R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *PLDI*, pages 196–207, 2003.
6. D. Bühler, P. Cuoq, B. Yakobowski, M. Lemerre, A. Maroneze, V. Perrelle, and V. Prevosto. *The EVA Plug-in*. CEA LIST, Software Reliability Laboratory, Saclay, France, F-91191, 2017. `https://frama-c.com/download/frama-c-value-analysis.pdf`.
7. A. Chawdhary and A. King. Compact Difference Bound Matrices. In *Asian Symposium on Programming Languages and Systems*, volume 10695 of *LNCS*, pages 471–490. Springer, 2017.
8. A. Chawdhary and A. King. Closing the Performance Gap between Doubles and Rationals for Octagons. Technical Report 67227, University of Kent, 2018. `https://kar.kent.ac.uk/67227/`.
9. A. Chawdhary, E. Robbins, and A. King. Simple and Efficient Algorithms for Octagons. In *APLAS*, volume 8858 of *LNCS*, pages 296–313. Springer, 2014.
10. A. Chawdhary, E. Robbins, and A. King. Incrementally Closing Octagons. *Formal Methods in System Design*, pages 1–46, 2018. `https://doi.org/10.1007/s10703-017-0314-7`.
11. D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *CAV*, volume 407 of *LNCS*, pages 187–212. Springer, 1989.
12. A. P. Ershov. On Programming of Arithmetic Operations. *Communications of the ACM*, 1(8):3–6, 1958. Translated by M. D. Friedman.
13. A. P. Ershov. Programming of Arithmetic Operations. *Doklady Akademii Nauk SSSR*, 118(3):427–430, 1958.
14. G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. Stuckey. Exploiting Sparsity in Difference-bound Matrices. In *SAS*, volume 9837 of *LNCS*, pages 189–211, 2016.
15. GNU. *GNU MP Manual*, 2016. `https://gmplib.org/manual/`.
16. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. Navas. The SeaHorn Verification Framework. In *CAV*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.
17. K. Heo, H. Oh, and H. Yang. Learning a Variable-Clustering Strategy for Octagon From Labeled Data Generated by a Static Analysis. In *SAS*, volume 9837 of *LNCS*, pages 237–256, 2016.
18. B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667, 2009.
19. J.-H. Jourdan. *Verasco: a Formally Verified C Static Analyzer*. PhD thesis, Université Paris Diderot (Paris 7) Sorbonne Paris Cité, May 2016.

20. D. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley, 1998.
21. J.-L. Lassez, T. Huynh, and K. McAloon. Simplication and Elimination of Redundant Linear Arithmetic Constraints. In *Constraint Logic Programming*, pages 73–87. MIT Press, 1993.
22. M. Measche and B. Berthomieu. Time Petri-nets for analyzing and verifying time dependent communication protocols. In H. Rudin and C. West, editors, *Protocol Specification, Testing and Verification III*, pages 161–172. North-Holland, 1983.
23. A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique En Informatique, 2004.
24. A. Miné. The Octagon Abstract Domain. *HOSC*, 19(1):31–100, 2006.
25. N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Trinity College, University of Cambridge, 2004.
26. H. Oh, L. Brutschy, and K. Yi. Access Analysis-Based Tight Localization of Abstract Memories. In *VMCAI*, volume 6538 of *LNCS*, pages 356–370, 2011.
27. H. Oh, K. Heo, W. Lee, D. Park, J. Kang, and K. Yi. Global Sparse Analysis Framework. *ACM TOPLAS*, 36(3):8:1–8:44, 2014.
28. H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective Context-Sensitivity Guided by Impact Pre-analysis. In *PLDI*, pages 475–484, 2014.
29. G. Singh, M. Püschel, and M. Vechev. Making Numerical Program Analysis Fast. In *PLDI*, pages 303–313. ACM Press, 2015.
30. C. Urban. FuncTion: An Abstract Domain Functor for Termination (Competition Contribution). In *TACAS*, volume 9035 of *LNCS*, pages 464–466. Springer, 2015.
31. A. Venet and G. Brat. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *PLDI*, pages 231–242, 2004.
32. L. F. Williams Jr. A Modification to the Half-Interval Search (Binary Search) Method. In *Proceedings of the 14th ACM Southeast Conference*, pages 95–101, 1976.